CROSS-MODEL REPRODUCTIVE FITNESS: EVOLVING TRANSFERABLE SKILLS ACROSS MODEL HOSTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents increasingly use reusable skills to perform long-horizon tasks involving planning, tool use, and artifact manipulation. Existing evolution protocols typically optimize skills on a primary host and evaluate transfer after optimization, even though skills can transfer across execution models and agent harnesses. This selection procedure can favor host-specific instructions and invocation patterns, and pooled performance can conceal degradation on weaker or incompatible hosts. We introduce Cross-Model Reproductive Fitness (CMRF), a transfer-aware evolutionary framework that models model–harness configurations as heterogeneous host environments and selects skills directly for cross-host utility. CMRF organizes hosts into evolution islands and evaluates candidate skills based on aggregate task gain, cross-host variability, weak-model gain, and worst-host performance. Skills migrate between islands and receive reproductive credit when they provide measurable utility in the destination host, aligning evolutionary survival with transferability. CMRF decomposes each skill into a platform-agnostic core, a model adapter, and a harness invocation layer, preserving shared procedures while supporting host-specific adaptation. Under matched execution budgets, we evaluate CMRF on seen and held-out host configurations against single-host evolution, pooled-average selection, worst-host-aware selection, and island migration. Ablation studies and structural analyses isolate the contributions of migration, transfer-aware fitness, and layered skill design to cross-host robustness.

1 INTRODUCTION

Large language model (LLM) agents increasingly address long-horizon tasks involving planning, tool use, artifact manipulation, and intermediate verification. An *agent skill* packages reusable procedural knowledge, executable routines, and supporting resources that augment an agent at inference time. Skills provide a modular mechanism for conveying specialized procedures across tasks without modifying model parameters. Their utility, however, depends on the execution environment. SkillsBench evaluates skills under multiple model–harness configurations and finds that performance gains vary substantially across execution models and agent harnesses(Li et al., 2026). We therefore treat a skill as a behavioral program whose effectiveness depends jointly on the skill, its host model, and its invocation interface.

Recent work automates the discovery and improvement of these programs. SkillWeaver discovers, practices, and distills reusable APIs for web agents, and reports that skills synthesized by stronger agents can improve weaker agents(Zheng et al., 2025). CoEvoSkills evolves structured, multi-file skill packages using a surrogate verifier and evaluates them on LLMs beyond the hosts used for construction(Zhang et al., 2026). These results establish the feasibility of cross-model transfer and motivate self-evolution as a scalable approach to skill creation. Existing methods typically assess transfer *after* optimization: they refine a skill under one primary model–harness configuration and then deploy it on other configurations to determine whether the learned behavior transfers across hosts.

This optimize-then-transfer protocol excludes host heterogeneity from the evolutionary dynamics. Selection under a single host can favor instructions, code structures, or invocation patterns tailored to that host's reasoning behavior and interface conventions. Optimizing pooled average performance does not fully address this issue because a high mean can conceal severe degradation on a weak

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

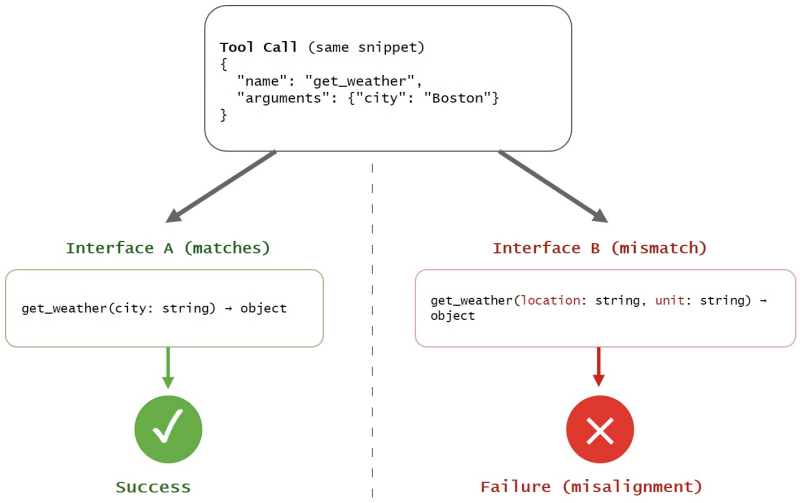

Figure 1: Overview of the CMRF evolutionary framework.

model or an incompatible harness. Such degradation is critical in heterogeneous deployments, where expensive models may author skills that smaller models execute, or where skills are shared across systems with different tool schemas and control loops. Addressing this setting requires *transfer-aware selection*: successful variants should retain utility across known hosts, avoid sacrificing the weakest host to improve average performance, and receive credit when they establish functionality in new hosts. The skill representation must also separate reusable procedural logic from model- and harness-specific adaptation so that host-specific components do not become entangled with the shared behavior.

We introduce **Cross-Model Reproductive Fitness (CMRF)**, an evolutionary framework that treats model–harness configurations as distinct host environments and directly selects for cross-host reproduction. As illustrated in 1, CMRF organizes hosts into evolution islands, each of which executes and refines a skill population using its own model and harness. The fitness function combines aggregate task gain, cross-host variability, weak-model gain, and worst-host performance, thereby penalizing candidates whose mean improvement depends on failures in specific hosts. Skills periodically migrate between islands, and a migrated skill receives *reproductive credit* only if it provides measurable utility in the destination host. Evolutionary survival therefore reflects transferable behavior instead of source-host specialization. CMRF represents each skill using three jointly evolvable layers: a *platform-agnostic core* that captures reusable procedural logic, a *model adapter* that accounts for model-specific execution characteristics, and a *harness invocation layer* that mediates tool and interface conventions. This representation preserves shared behavior while supporting localized adaptation to new hosts.

We evaluate CMRF by evolving skills across combinations of execution models and agent harnesses and testing them on held-out host configurations. Under matched execution budgets, we compare single-host evolution, pooled-average selection, worst-host-aware selection, and island migration. We further examine whether transferable skills provide larger gains to weaker models and analyze how linguistic structure, executable modules, and verification routines relate to transfer. Our ablations remove inter-island migration, the worst-host term, and the layered representation. Additional controls vary the migration frequency and isolate changes in the execution model from changes in the harness. These experiments separate cross-host adaptation from gains attributable to increased sampling or post hoc transfer.

Our contributions are:

- We formulate agent-skill evolution as multi-host optimization, where models and harnesses define heterogeneous execution environments, and show why post-evolution evaluation alone does not incorporate cross-model transfer into skill selection.

- We propose CMRF, a transfer-aware evolutionary framework that combines cross-host utility, variability, weak-model gain, and worst-host performance with destination-based reproductive credit.
- We develop an island migration mechanism and a layered skill representation that separates platform-agnostic procedural knowledge, model adaptation, and harness invocation, supporting controlled specialization while preserving a shared transferable core.
- We define a budget-matched evaluation protocol spanning seen and unseen model–harness configurations, together with ablations and structural analyses that test whether transfer-oriented selection improves robustness beyond single-host evolution and pooled-average optimization.

2 Related Work

2.1 Procedural Knowledge, Tool Use, and Skill Acquisition

LLM agents combine reasoning with action interfaces, persistent state, and executable procedures. ReAct interleaves reasoning and environment actions(Yao et al., 2022), while SayCan grounds model proposals in robotic affordances(Ahn et al., 2022). Toolformer learns API use from self-supervised annotations(Schick et al., 2023), whereas Gorilla and ToolLLM support tool selection and argument generation through instruction tuning, documentation, retrieval, and execution trajectories(Patil et al., 2023)(Qin et al., 2023). API-Bank evaluates executable API use(Li et al., 2023), WebGPT learns browser-assisted question answering(Nakano et al., 2021), HuggingGPT delegates tasks to specialized models(Shen et al., 2023), and ART retrieves demonstrations to construct tool-using programs(Paranjape et al., 2023).

Reusable skills extend tool calls into persistent procedures. Voyager builds an executable program library(Wang et al., 2023), LLMs as Tool Makers separates function construction from invocation(Cai et al., 2023), and SkillWeaver distills web interactions into transferable skills(Zheng et al., 2025). CoEvoSkills evolves structured skill packages and surrogate verifiers(Zhang et al., 2026), while SkillsBench shows that effectiveness varies across skills, models, and agent harnesses(Li et al., 2026).

Memory-based adaptation instead retains interaction experience. Reflexion stores verbal feedback(Shinn et al., 2023), Self-Refine iterates self-feedback and revision(Madaan et al., 2023), Generative Agents retrieve synthesized reflections(Park et al., 2023), and MemGPT manages information across memory tiers(Packer et al., 2023). Unlike executable skills, such memories often encode model-dependent diagnoses or heuristics.

2.2 Optimization and Evolution of Language-Model Programs

Prompt optimization treats instructions as discrete programs evaluated by downstream performance. Automatic Prompt Engineer generates and selects instructions(Zhou et al., 2022), Automatic Prompt Optimization revises prompts using error-derived critiques(Pryzant et al., 2023), and OPRO conditions optimization on evaluated solutions and scores(Yang et al., 2023). Population-based methods expand search through LLM-mediated evolution: EvoPrompt applies genetic and differential-evolution operators(Guo et al., 2023), Promptbreeder co-evolves task and mutation prompts(Fernando et al., 2023), and Self-Taught Optimizer recursively improves solution-generating code(Zelikman et al., 2023).

Optimization also targets compound programs. DSPy compiles modular prompting pipelines(Khattab et al., 2023), MIPRO jointly optimizes instructions and demonstrations(Opsahl-Ong et al., 2024), and TextGrad propagates natural-language feedback through computation graphs(Yuksekgonul et al., 2024). Multi-agent frameworks further encode behavior through roles and protocols: AutoGen uses configurable agent conversations(Wu et al., 2023), AgentVerse separates collaboration into staged processes(Chen et al., 2023), MetaGPT implements role-specific operating procedures(Hong et al., 2023), and ChatDev uses staged dialogues for software production(Qian et al., 2023). These systems make runtime and harness dependence explicit.

2.3 Agent Evaluation, Generalization, and Host Heterogeneity

Interactive benchmarks evaluate end-to-end behavior. AgentBench spans diverse environments(Liu et al., 2023), WebArena measures functional completion on reproducible websites(Zhou et al., 2023), GAIA evaluates assistants requiring reasoning and tool use(Mialon et al., 2023), SWE-bench tests repository-level repair(Jimenez et al., 2023), and OSWorld evaluates multimodal desktop interaction(Xie et al., 2024).

Generalization includes transfer to unseen tools(Patil et al., 2023)(Qin et al., 2023), tasks and environments(Wang et al., 2023)(Zheng et al., 2025), and domains or modalities(Liu et al., 2023)(Xie et al., 2024). Cross-model evidence indicates that stronger models can produce procedures useful to weaker ones, although gains depend on the receiving model and harness(Cai et al., 2023)(Zheng et al., 2025)(Li et al., 2026).

Most optimization scalarizes performance, potentially hiding variation across configurations. Evolutionary methods preserve candidate diversity(Guo et al., 2023)(Fernando et al., 2023), compound optimizers modify interacting modules(Opsahl-Ong et al., 2024)(Yuksekgonul et al., 2024), and skill systems permit evaluation across agents(Zhang et al., 2026)(Li et al., 2026).

3 Method

We introduce Cross-Model Reproductive Fitness (CMRF), which represents each model–harness configuration as a host and incorporates cross-host utility into evolutionary selection. CMRF favors skills that improve their source hosts, remain stable across heterogeneous hosts, and retain positive utility after migration.

3.1 Problem Formulation and Framework

Let $\mathcal{H}_{\mathrm{src}} = \{h_i\}_{i=1}^{H}$ denote the source hosts, where each $h_i = (m_i, a_i)$ comprises an execution model m_i and a harness a_i. The harness defines the interaction protocol, tool interface, skill-loading mechanism, and execution environment. Each host is associated with a task distribution $\mathcal{D}_{h_i}$. For $x \sim \mathcal{D}_{h_i}$ and an optional skill $\mathbf{s}$, a bounded verifier $q(h_i, x, \mathbf{s}) \in [0, 1]$ measures execution quality; $\varnothing$ denotes execution without an external skill.

For an evaluation set $\mathcal{E}_{h_i} \subset \mathcal{D}_{h_i}$, the host-specific gain is

$$g_{h_i}(\mathbf{s}) = \frac{1}{|\mathcal{E}_{h_i}|} \sum_{x \in \mathcal{E}_{h_i}} [q(h_i, x, \mathbf{s}) - q(h_i, x, \varnothing)]. \tag{1}$$

This paired difference separates skill utility from the host's unassisted capability. CMRF evolves skills with positive and stable gains on source hosts and evaluates their utility on disjoint probe hosts $\mathcal{H}_{\mathrm{probe}}$.

CMRF maintains one island $\mathcal{I}_h$ for each source host. Each island evolves and evaluates a population using tasks from $\mathcal{D}_h$. At scheduled rounds, selected candidates undergo cross-host evaluation and migrate to other configurations. Migration preserves reusable content while adapting host-specific interfaces, and reproductive credit is determined by the realized gain on each destination host. Figure 2 illustrates this host-island evolution, cross-host evaluation, and migration framework. The complete optimization procedure is provided in the Appendix.

3.2 Structured Skill Genome

A skill instantiated for host h has the genome $\mathbf{s}_h = (\mathbf{c}, \mathbf{d}_h, \mathbf{v}_h)$, where $\mathbf{c}$ is a platform-agnostic core, $\mathbf{d}_h$ is a model adapter, and $\mathbf{v}_h$ is a harness invocation layer. The core specifies task decomposition, reusable procedures, error checks, and validation rules without host-specific commands, wrappers, paths, or tool schemas. The adapter converts these procedures into instructions for m_h, while the invocation layer maps abstract operations to the loading and tool-calling conventions of a_h.

Layer-wise mutation produces

$$\widetilde{\mathbf{s}}_h = \mathcal{M}_\ell(\mathbf{s}_h; \mathcal{B}_h, \boldsymbol{\xi}), \qquad \ell \in \{\mathrm{core}, \mathrm{adapter}, \mathrm{invocation}\}, \tag{2}$$

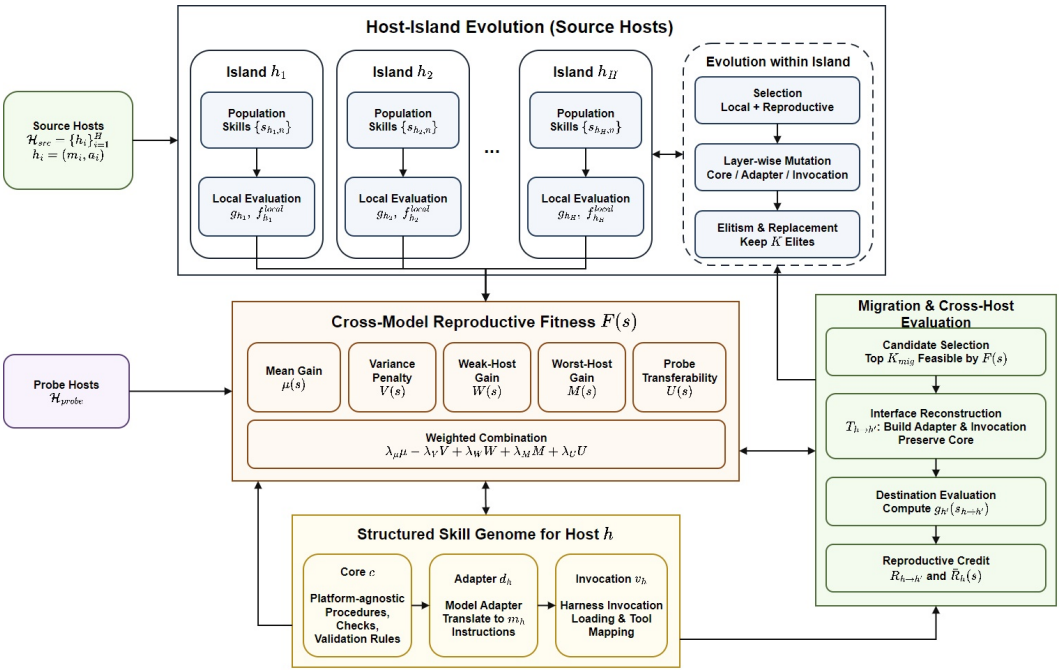

Figure 2: Overview of the CMRF framework.

where $\mathcal{B}_h$ denotes accumulated execution feedback and $\boldsymbol{\xi}$ denotes mutation randomness. Core mutations revise general procedures, whereas interface mutations repair host-local incompatibilities. By default, migration preserves the core and selects or regenerates the destination-specific interface layers.

3.3 Host-Island Evolution

Island $\mathcal{I}_h$ maintains a population $\mathcal{P}_h^{(t)} = \{\mathbf{s}_{h,n}^{(t)}\}_{n=1}^{N}$. At generation t, the island samples parents according to local and reproductive fitness, applies Eq. (2), and evaluates the resulting offspring on $\mathcal{B}_h^{(t)} \subset \mathcal{D}_h$. The local fitness is

$$f_h^{\text{local}}(\mathbf{s}) = \frac{1}{|\mathcal{B}_h^{(t)}|} \sum_{x \in \mathcal{B}_h^{(t)}} [q(h, x, \mathbf{s}) - q(h, x, \varnothing)] - \lambda_{\text{cost}} c(h, \mathbf{s}), \tag{3}$$

where $c(h, \mathbf{s})$ is the normalized execution cost and $\lambda_{\text{cost}} \geq 0$. The cost term is applied only when candidate budgets differ; setting $\lambda_{\text{cost}} = 0$ yields gain-only selection. Each island retains K elites and fills the remaining positions with offspring and migrants. Cross-host reproductive fitness further determines long-term survival.

3.4 Cross-Model Reproductive Fitness

Given host weights $w_h \geq 0$ satisfying $\sum_h w_h = 1$, the average cross-host gain and its dispersion are $\mu(\mathbf{s}) = \sum_{h \in \mathcal{H}_{\text{src}}} w_h g_h(\mathbf{s})$, and $V(\mathbf{s}) = \sum_{h \in \mathcal{H}_{\text{src}}} w_h [g_h(\mathbf{s}) - \mu(\mathbf{s})]^2$. Uniform weights assign equal importance to all hosts, while predefined nonuniform weights can represent deployment frequency. The variance penalty limits host sensitivity, although individual hosts can still exhibit negative gains.

Let the unassisted capability of host h be

$$b_h = \frac{1}{|\mathcal{E}_h|} \sum_{x \in \mathcal{E}_h} q(h, x, \varnothing). \tag{4}$$

Define $\mathcal{H}_{\text{weak}}$ as the bottom ρ-quantile of hosts ranked by b_h, where $\rho \in (0, 1]$ is fixed. The weak-host and worst-host gains are

$$W(\mathbf{s}) = \frac{1}{|\mathcal{H}_{\text{weak}}|} \sum_{h \in \mathcal{H}_{\text{weak}}} g_h(\mathbf{s}), \qquad M(\mathbf{s}) = \min_{h \in \mathcal{H}_{\text{src}}} g_h(\mathbf{s}) \tag{5}$$

Probe hosts generate neither mutations nor parents. Transferring a skill from host h to destination h' produces $T_{h \to h'}(\mathbf{s}_h) = (\mathbf{c}, \mathcal{A}_{m_{h'}}(\mathbf{c}, \mathbf{d}_h), \mathcal{V}_{a_{h'}}(\mathbf{c}, \mathbf{v}_h))$, where $\mathcal{A}_{m_{h'}}$ and $\mathcal{V}_{a_{h'}}$ construct destination-specific interfaces while preserving the core. Probe transferability is

$$U(\mathbf{s}) = \frac{1}{|\mathcal{H}_{\text{probe}}|} \sum_{h' \in \mathcal{H}_{\text{probe}}} g_{h'}(T_{h \to h'}(\mathbf{s})). \tag{6}$$

The combined reproductive fitness is

$$F(\mathbf{s}) = \lambda_\mu \mu(\mathbf{s}) - \lambda_V V(\mathbf{s}) + \lambda_W W(\mathbf{s}) + \lambda_M M(\mathbf{s}) + \lambda_U U(\mathbf{s}), \tag{7}$$

where the nonnegative coefficients are fixed using development tasks. The five terms measure average utility, cross-host stability, weak-host gain, worst-host gain, and probe transferability, respectively. This score serves as a selection criterion and is not optimized as a differentiable loss. Feasible candidates can additionally be required to satisfy $g_h(\mathbf{s}) \geq -\epsilon$ for all $h \in \mathcal{H}_{\text{src}}$, where $\epsilon \geq 0$ is predefined. During migration and elite selection, feasible candidates rank above infeasible candidates.

3.5 Migration, Reproductive Credit, and Deployment

Every τ generations, each island selects up to K_{mig} feasible candidates according to reproductive fitness. For each $h' \neq h$, migration constructs $\mathbf{s}_{h \to h'} = T_{h \to h'}(\mathbf{s}_h)$, and the destination evaluates the migrated skill on a held-out minibatch. Its realized reproductive credit is

$$R_{h \to h'}(\mathbf{s}) = g_{h'}(\mathbf{s}_{h \to h'}) - \max\{0, \overline{g}_{h'}^{(t)}\}, \tag{8}$$

where $\overline{g}_{h'}^{(t)}$ is the destination population's mean gain before migration. Positive credit therefore requires the migrated skill to exceed both the no-skill baseline and the current destination reference. The cumulative credit is

$$\overline{R}_h(\mathbf{s}) = \frac{1}{|\mathcal{M}_h(\mathbf{s})|} \sum_{h' \in \mathcal{M}_h(\mathbf{s})} R_{h \to h'}(\mathbf{s}), \tag{9}$$

where $\mathcal{M}_h(\mathbf{s})$ is the set of visited destinations. Parent selection on the source island combines $F(\mathbf{s})$ with $\overline{R}_h(\mathbf{s})$, while migrants on the destination island compete with local offspring according to Eq. (3).

Evolution returns a feasible archive $\mathcal{Q}$ whose members are nondominated with respect to $(\mu, -V, W, M, U)$. For an unseen host $h_{\text{new}} = (m_{\text{new}}, a_{\text{new}})$, the method selects a canonical core $\mathbf{c}^\star$ using deployment weights or a local validation subset and instantiates $\mathbf{s}_{h_{\text{new}}} = (\mathbf{c}^\star, \mathcal{A}_{m_{\text{new}}}(\mathbf{c}^\star), \mathcal{V}_{a_{\text{new}}}(\mathbf{c}^\star))$. When limited feedback is available, refinement is restricted to the adapter and invocation layers, preserving the transferable core.

4 Experiments

We evaluate whether transfer-aware evolution produces effective skills across heterogeneous model–harness hosts. This design tests whether gains arise from reusable procedures rather than favorable assumptions about a particular execution environment.

4.1 Evaluation Protocol

We evaluate all methods on 48 verifier-backed agent tasks. The task set spans distinct interaction patterns and requires each final answer to satisfy the same deterministic verification criteria. Each host pairs an execution model with an agent harness. The four source hosts combine a stronger model $\mathcal{M}_{\text{s}}$ and a weaker model $\mathcal{M}_{\text{w}}$ with two harnesses, $\mathcal{H}_{\text{a}}$ and $\mathcal{H}_{\text{b}}$.

Table 1: Performance on the four source model–harness hosts. Values are mean±standard deviation over five evolutionary seeds. Host Std. measures dispersion among host-level pass rates, so lower is better. All evolutionary methods use the same candidate-evaluation budget.

Method	Mean Pass Rate ↑	Weak Host ↑	Worst Host ↑	Host Std. ↓
No Skill	31.9±0.8	22.6±1.1	22.6±1.1	8.9±0.5
Best-of-N	40.8±1.3	30.5±1.8	29.9±1.6	8.2±0.8
Single-Host Evolution	43.7±1.1	31.8±1.5	30.9±1.4	8.7±0.7
Worst-Case Selection	44.9±1.0	37.9±1.3	37.1±1.2	5.6±0.6
Pareto Search	44.5±1.2	34.6±1.4	33.8±1.5	6.9±0.7
Pooled Average	45.8±0.9	36.4±1.2	35.8±1.1	7.1±0.6
CMRF (ours)	**48.6±0.8**	**41.5±1.0**	**40.7±1.0**	**4.8±0.4**

Table 2: Transfer to four unseen model–harness configurations without task-level skill refinement. "Both Unseen" changes both the execution model and harness. Values are mean±standard deviation over five seeds.

Method	New Model A	New Model B	New Harness	Both Unseen	Average
No Skill	29.8±1.0	25.7±1.2	30.9±1.1	23.6±1.3	27.5±0.9
Best-of-N	37.2±1.5	33.1±1.6	36.4±1.4	29.7±1.8	34.1±1.3
Single-Host Evolution	38.9±1.4	34.8±1.5	36.7±1.6	28.7±1.9	34.8±1.4
Worst-Case Selection	41.5±1.1	38.2±1.3	40.4±1.2	35.1±1.5	38.8±1.0
Pooled Average	42.7±1.0	38.4±1.2	40.8±1.1	33.8±1.6	38.9±0.9
CMRF (ours)	**45.1±0.9**	**42.0±1.0**	**44.3±0.9**	**38.4±1.3**	**42.5±0.8**

We measure transfer on four unseen configurations: two new models paired with a source harness, a source model paired with a new harness, and a configuration with both components unseen. We freeze the skills before transfer and supply only prescribed adapter files, without task-specific edits or further evolution. This restriction isolates zero-shot portability and prevents transfer results from benefiting from additional search, manual correction, or hidden evaluation feedback.

We compare CMRF with No Skill, Best-of-N, Single-Host Evolution on $(\mathcal{M}_{\mathrm{s}}, \mathcal{H}_{\mathrm{a}})$, Pooled Average selection, worst-host optimization, and Pareto Search over mean performance and execution cost. The primary metric is verifier-backed task pass rate. We also report weak-host and minimum-host pass rates and host-level standard deviation. Results are mean±standard deviation over five paired evolutionary seeds.

Each method receives 768 candidate evaluations under a fixed budget. CMRF maintains four host-specific islands. Selection considers mean gain, weak-host gain, worst-host performance, cross-host dispersion, and successful reproduction on a destination host. Each skill separates transferable procedural logic from model adaptation and harness invocation.

4.2 Source-Host Performance

Table 1 reports performance on the four source hosts. CMRF achieves a mean pass rate of 48.6%, exceeding Pooled Average by 2.8 percentage points and Single-Host Evolution by 4.9 points. Compared with Pooled Average, CMRF increases the minimum-host pass rate from 35.8% to 40.7% and reduces host-level dispersion from 7.1 to 4.8 points.

Single-Host Evolution nearly matches CMRF on its optimization host but trails it by 9.8 points on the worst source host, indicating specialization. Worst-Case Selection raises the minimum pass rate to 37.1%, but its mean remains 3.7 points below that of CMRF.

The improvement is larger under difficult transfer conditions. Compared with Pooled Average, CMRF improves by 1.4 points on $(\mathcal{M}_{\mathrm{s}}, \mathcal{H}_{\mathrm{a}})$ and 5.1 points on $(\mathcal{M}_{\mathrm{w}}, \mathcal{H}_{\mathrm{b}})$.

Table 3: Ablations and efficiency under the same task suite. Source Mean and Unseen Worst report pass rate; Host Std. is lower-is-better. Cost is the total normalized execution cost relative to a single-host run.

Variant	Source Mean ↑	Unseen Worst ↑	Host Std. ↓	Cost ↓
CMRF (full)	**48.6±0.8**	**38.4±1.3**	**4.8±0.4**	2.71×
w/o island migration	46.2±1.0	34.6±1.5	6.5±0.6	2.63×
w/o worst-host term	47.1±0.9	34.9±1.6	6.7±0.7	2.68×
w/o variance penalty	47.8±0.9	36.1±1.4	6.3±0.6	2.69×
w/o weak-host credit	46.9±1.0	35.5±1.5	5.9±0.5	2.67×
monolithic skill	45.7±1.1	33.8±1.7	7.0±0.7	2.54×
migration every generation	47.3±1.0	36.7±1.4	5.5±0.5	2.92×

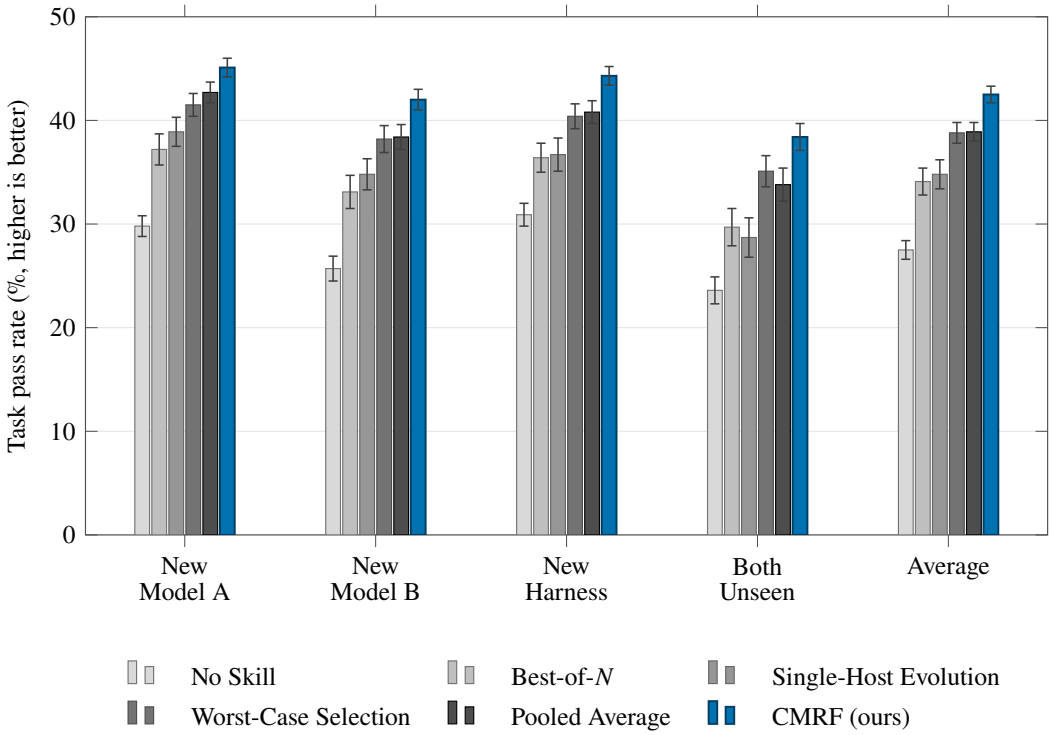

Figure 3: Transfer performance of frozen skills on four unseen model–harness configurations without task-level refinement. Bars report mean verifier-backed pass rate over five paired evolutionary seeds; error bars show standard deviation.

4.3 Generalization Beyond the Evolution Hosts

Table 2 evaluates frozen skills on unseen hosts, and Figure 3 shows the four configurations. CMRF achieves the highest pass rate in all conditions, averaging 42.5%. This exceeds Pooled Average by 3.6 points and Single-Host Evolution by 7.7 points. When both model and harness change, CMRF achieves 38.4%, compared with 33.8% and 28.7%, respectively. The decrease from source-host mean to unseen-host average is 6.1 points for CMRF, 6.9 for Pooled Average, and 8.9 for Single-Host Evolution.

The weaker unseen model improves from 25.7% without a skill to 42.0% with CMRF, a normalized gain of 63.4%. The stronger unseen model improves from 29.8% to 45.1%, a normalized gain of 51.3%.

Transfer remains lower when both host components change. The 38.4% pass rate is 4.1 points below the unseen-host average. Residual errors persist. Layered adaptation reduces, but does not eliminate, coupling between procedural decisions and invocation semantics.

4.4 Mechanism Ablations

Table 3 isolates components supporting robust transfer. Removing migration reduces the source mean from 48.6% to 46.2% and the unseen worst-host score from 38.4% to 34.6%. Removing the worst-host term lowers the unseen minimum by 3.5 points, while removing weak-host reproductive credit lowers it by 2.9 points. Removing the variance penalty increases host dispersion from 4.8 to 6.3 points.

The layered representation yields the largest individual transfer improvement. Replacing the core–adapter–invocation decomposition with a monolithic skill reduces the source mean by 2.9 points and the unseen worst-host score by 4.6 points. Monolithic mutations can repair host-specific calls by modifying shared logic, causing regressions on previously successful hosts. Layering confines model-facing and harness-specific changes while preserving transferable checks and executable modules.

Migration frequency affects transfer and cost. Migrating every generation reduces the unseen worst-host score by 1.7 points and increases cost from $2.71\times$ to $2.92\times$. Frequent migration spreads high-scoring cores before local adapters stabilize and reduces island diversity, while no migration permits specialization. Robust transfer therefore depends on heterogeneous selection pressure and localized repair.

4.5 Efficiency

CMRF costs $2.71\times$ as much as a single-host evolutionary run because confirmation includes hosts with different execution costs. Proxy pre-screening rejects 58% of mutations before full evaluation, reducing normalized cost from $4.36\times$ to $2.71\times$ while retaining the 768-evaluation budget. A run requires 29 GPU-hours and 8.4 wall-clock hours with parallel island evaluation. At deployment, skills add no model parameters and increase median latency by 3.8%. Compared with Single-Host Evolution, this cost increases source-host mean pass rate by 4.9 points and unseen-host average by 7.7 points, while reducing source-host dispersion from 8.7 to 4.8 points.

5 Conclusion

This paper presented Cross-Model Reproductive Fitness (CMRF), a transfer-aware evolutionary framework that integrates heterogeneous model–harness environments into skill selection. Its island migration mechanism, destination-based reproductive credit, multi-host fitness function, and layered skill representation promote transferable procedural behavior while retaining host-specific adaptation. Budget-matched evaluations across seen and held-out host configurations established CMRF's cross-host effectiveness relative to alternative selection strategies. Ablations and structural analyses further verified the contributions of migration, worst-host-aware selection, and layered skill organization, confirming that CMRF produces robust skills whose utility extends across execution models and agent harnesses.

Limitations

CMRF relies on scalarized fitness objectives whose weighting may not fully capture all practical notions of transferability. The layered skill representation also assumes that platform-agnostic procedures can be cleanly separated from model- and harness-specific behavior, whereas these components may interact in more complex deployments. Moreover, the evolutionary process introduces additional computational and implementation overhead compared with simpler optimization strategies. Future work could explore adaptive fitness weighting, richer representations of host interactions, and further experiments across broader application domains.

DISCLOSURE OF AI TOOL USAGE

The authors wish to disclose that AI-powered writing assistants were used in a limited and clearly defined scope during the drafting of this manuscript. Their application was confined to language editing tasks, such as improving the fluency of written passages, correcting grammatical mistakes, and standardizing terminology across sections. All substantive research contributions—including the cross-model reproductive fitness formulation, the transferable skill evolution framework, the multi-host evaluation protocol, and the analysis of empirical results—were produced exclusively by the human authors. No AI tool was involved in generating research hypotheses, designing experiments, or interpreting findings. Every AI-assisted modification was reviewed and validated by the authors before inclusion in the final text. The authors assume full accountability for all scientific content in this paper.

REFERENCES

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022. URL https://arxiv.org/abs/2204.01691.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. *arXiv preprint arXiv:2305.17126*, 2023. URL https://arxiv.org/abs/2305.17126.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. *arXiv preprint arXiv:2308.10848*, 2023. URL https://arxiv.org/abs/2308.10848.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023. URL https://arxiv.org/abs/2309.16797.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023. URL https://arxiv.org/abs/2309.08532.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023. URL https://arxiv.org/abs/2308.00352.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023. URL https://arxiv.org/abs/2310.03714.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023. URL https://arxiv.org/abs/2304.08244.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, Chujun Tao, Binxu Li, Xuandong Zhao, Hejia Geng, Xiaojun Wu, Junwei Zhou, Xiaokun Chen, Hanwen Xing, Yubo Li, Qunhong Zeng, Di Wang, Yuanli Wang, Roey Ben Chaim, Penghao Jiang, Haotian Shen, Luyang Kong, Xinyi Liu, Runhui Wang, Xuanqing Liu, Jiachen Li, Xin Lan, Yueqian Lin, Wengao Ye, Junwei He, Songlin Li, Yue Zhang, Yipeng Gao, Yijiang Li, Ze Ma, Liqiang Jing, Tianyu Wang, Kaixin Li, Yiqi Xue, Haoran Lyu, Yizhuo He, Yuchen Tian, Shutong Wu, Bowei Wang, Yixuan Gao, Bo Chen, Litong Liu, Sikai Cheng, Jiajun Bao, Shuaicheng Tong, Shuwen Xu, Terry Yue Zhuo, Tinghan Ye, Qi Qi, Miao Li, Longtai Liao, Zelin Tan, Chang Shi, Xilin Tang, Srinath Tankasala, Boqin Yuan, Yaoyao Qian, Jianhong Tu, Chenguang Wang, Yizhou Sun, Wei Wang, Aaron Taylor, Ziyue Yang, Changkun Guan, Zhikang Dong, Xinyu Zhang, Steven Dillmann, Han chung Lee, and Dawn Song. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026. URL https://arxiv.org/abs/2602.12670.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023. URL https://arxiv.org/abs/2308.03688.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023. URL https://arxiv.org/abs/2303.17651.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. *arXiv preprint arXiv:2311.12983*, 2023. URL https://arxiv.org/abs/2311.12983.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021. URL https://arxiv.org/abs/2112.09332.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. *arXiv preprint arXiv:2406.11695*, 2024. URL https://arxiv.org/abs/2406.11695.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023. URL https://arxiv.org/abs/2310.08560.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023. URL https://arxiv.org/abs/2303.09014.

Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023. URL https://arxiv.org/abs/2304.03442.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023. URL https://arxiv.org/abs/2305.15334.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023. URL https://arxiv.org/abs/2305.03495.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev:

Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023. URL https://arxiv.org/abs/2307.07924.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023. URL https://arxiv.org/abs/2307.16789.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023. URL https://arxiv.org/abs/2302.04761.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *arXiv preprint arXiv:2303.17580*, 2023. URL https://arxiv.org/abs/2303.17580.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023. URL https://arxiv.org/abs/2303.11366.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023. URL https://arxiv.org/abs/2308.08155.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024. URL https://arxiv.org/abs/2404.07972.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023. URL https://arxiv.org/abs/2309.03409.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/abs/2210.03629.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024. URL https://arxiv.org/abs/2406.07496.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023. URL https://arxiv.org/abs/2310.02304.

Hanrong Zhang, Shicheng Fan, Henry Peng Zou, Yankai Chen, Zhenting Wang, Jiayu Zhou, Chengze Li, Wei-Chieh Huang, Yifei Yao, Kening Zheng, Xue (Steve) Liu, Xiaoxiao Li, and Philip S. Yu. Coevoskills: Self-evolving agent skills via co-evolutionary verification. *arXiv preprint arXiv:2604.01687*, 2026. URL https://arxiv.org/abs/2604.01687.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, and Yu Su. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025. URL https://arxiv.org/abs/2504.07079.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023. URL https://arxiv.org/abs/2307.13854.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022. URL https://arxiv.org/abs/2211.01910.

APPENDIX

A CROSS-HOST EVOLUTION, MIGRATION, AND ARCHIVE CONSTRUCTION

This section specifies how CMRF integrates host-local mutation, cross-host evaluation, migration, reproductive credit, and population replacement. CMRF uses selection-based optimization: the layer-wise mutation operator modifies genomes, and verifier-derived fitness values determine candidate rankings.

A.1 EVALUATION AND SELECTION SEMANTICS

For a candidate originating from source host h, let

$$\mathbf{g}(\mathbf{s}) = \big(g_{h_1}(\mathbf{s}), \ldots, g_{h_H}(\mathbf{s})\big) \in \mathbb{R}^H$$

denote its source-host gain vector. Evaluation on host h_i first instantiates the candidate for that host. For a candidate $\mathbf{s}_h = (\mathbf{c}, \mathbf{d}_h, \mathbf{v}_h)$ originating from h, its gain on h_i is evaluated using

$$\mathbf{s}_{h \to h_i} = T_{h \to h_i}(\mathbf{s}_h) = \Big(\mathbf{c}, \mathcal{A}_{m_{h_i}}(\mathbf{c}, \mathbf{d}_h), \mathcal{V}_{a_{h_i}}(\mathbf{c}, \mathbf{v}_h)\Big).$$

When $h_i = h$, evaluation uses the host-local genome $\mathbf{s}_h$ directly. Each component of $\mathbf{g}(\mathbf{s})$ therefore measures the utility obtained through the interface of the evaluated host, while transfer preserves the platform-agnostic core.

The quantities $\mu(\mathbf{s})$, $V(\mathbf{s})$, $W(\mathbf{s})$, and $M(\mathbf{s})$ are computed from $\mathbf{g}(\mathbf{s})$ over the source hosts. Probe transferability $U(\mathbf{s})$ is computed analogously by constructing a probe-specific adapter and invocation layer for each $h' \in \mathcal{H}_{\text{probe}}$. Probe hosts provide verifier evaluations only: they maintain no populations, generate no offspring, and supply no parents.

Define the source-host feasibility indicator

$$\phi(\mathbf{s}) = \mathbb{I}\left[g_{h_i}(\mathbf{s}) \geq -\epsilon \quad \text{for every } i \in \{1, \ldots, H\}\right]. \tag{10}$$

When feasibility is enforced, selection orders candidates lexicographically by ϕ: every candidate with $\phi(\mathbf{s}) = 1$ ranks above every candidate with $\phi(\mathbf{s}) = 0$. Reproductive fitness $F(\mathbf{s})$ orders candidates within the feasible set during cross-host selection and migration. On a destination island, the instantiated migrant is evaluated and competes with local offspring according to f_h^{local}. Once a source candidate has visited at least one destination, parent selection also uses its cumulative reproductive credit $\overline{R}_h(\mathbf{s})$. This credit is a verifier-derived selection signal with no differentiable objective.

A.2 MIGRATION AND CREDIT PROPAGATION

Suppose that $\mathbf{s}_h$ migrates from source island $\mathcal{I}_h$ to destination island $\mathcal{I}_{h'}$. Before inserting the migrant, CMRF computes the destination reference from the destination population:

$$\overline{g}_{h'}^{(t)} = \frac{1}{|\mathcal{P}_{h'}^{(t)}|} \sum_{\mathbf{u} \in \mathcal{P}_{h'}^{(t)}} g_{h'}(\mathbf{u}). \tag{11}$$

The transferred candidate $\mathbf{s}_{h \to h'}$ is then evaluated on the held-out destination minibatch. The resulting credit is assigned to the source candidate:

$$R_{h \to h'}(\mathbf{s}_h) = g_{h'}(\mathbf{s}_{h \to h'}) - \max\{0, \overline{g}_{h'}^{(t)}\}.$$

After visiting the nonempty destination set $\mathcal{M}_h(\mathbf{s}_h)$, the source-side credit is computed as

$$\overline{R}_h(\mathbf{s}_h) = \frac{1}{|\mathcal{M}_h(\mathbf{s}_h)|} \sum_{h' \in \mathcal{M}_h(\mathbf{s}_h)} R_{h \to h'}(\mathbf{s}_h).$$

Interface reconstruction affects this credit through the realized destination gain. The credit remains attached to the source genome that supplied the transferred core.

A.3 Complete CMRF Procedure

Let T denote the predefined number of generations. For each island, $\mathcal{C}_h^{(t)}$ denotes the offspring generated at generation t, and $\mathcal{G}_h^{(t)}$ denotes the migrants received by host h during that generation. The population update retains K local elites and fills the remaining positions with evaluated offspring and received migrants. Selection induces the replacement $\mathcal{P}_h^{(t)} \mapsto \mathcal{P}_h^{(t+1)}$ without gradient updates.

The procedure terminates after the predefined T generations. The returned archive contains the feasible candidates that are nondominated under the five criteria defining cross-host reproductive fitness. To deploy a candidate on an unseen host, a canonical core is selected from the archive using the stated deployment weights or a local validation subset. The adapter and invocation layers are then constructed for the unseen host. When limited host feedback is available for refinement, mutation is restricted to these two interface layers and leaves the selected core unchanged.

B Environment Details

B.1 Datasets and Task Settings

We evaluate all methods on a verifier-backed suite of 48 long-horizon agent tasks spanning software engineering, data analysis, document processing, and structured web interaction. The software-engineering tasks require agents to edit multi-file repositories and pass unit tests. The data-analysis tasks require transformations of tabular data and the production of artifacts that satisfy schema- and value-level constraints. The document-processing tasks require agents to extract, revise, or generate structured office documents. The web-interaction tasks require agents to complete stateful workflows through predefined browser tools. Successful reference trajectories contain between 4 and 18 tool-mediated actions, with a median of 9.

We partition the suite into 24 evolution tasks, 8 development tasks, and 16 test tasks. The evolution tasks provide execution feedback for mutation and local selection. The development split determines the fitness coefficients, migration frequency, and deployment-time archive selection. We access the test split only for final reporting. All methods use the same split, task instructions, available tools, initial environment states, context limits, and execution budgets. To prevent task leakage, we hide task identifiers and test verifiers from the mutation model, exclude test-task trajectories from mutation feedback, and remove repository- or document-specific strings shared across splits using exact-match and MinHash-based near-duplicate filtering.

Each host pairs an execution model with an agent harness. The four source hosts form the Cartesian product of two execution models, Qwen2.5-72B-Instruct and Qwen2.5-7B-Instruct, and two harnesses: a ReAct-style sequential tool loop and a plan–execute–verify loop with explicit artifact inspection. We determine model strength from no-skill development performance, which identifies Qwen2.5-72B-Instruct as the strong source model and Qwen2.5-7B-Instruct as the weak source model. We evaluate transfer on four frozen-skill probe configurations: Llama-3.1-70B-Instruct with the source ReAct harness, Mistral-7B-Instruct-v0.3 with the source ReAct harness, Qwen2.5-72B-Instruct with an unseen function-calling harness, and Llama-3.1-70B-Instruct with the unseen harness. During evolution, probe hosts do not generate mutations, provide parent candidates, or expose task-level feedback.

Each task consists of a natural-language goal, tool schemas, workspace metadata, and explicit artifact constraints. Agents can reason internally and issue structured tool calls, and they must terminate with a final status object that contains the produced artifact paths and a concise completion report. Each episode is limited to 32 model turns, 24 tool calls, 32,768 input tokens, and 8,192 generated tokens. An episode terminates early upon verifier-confirmed success, an unrecoverable tool error,

Algorithm 1 Cross-Model Reproductive Fitness Evolution

Require: Source hosts $\mathcal{H}_{\mathrm{src}}$; probe hosts $\mathcal{H}_{\mathrm{probe}}$; initial populations $\{\mathcal{P}_h^{(0)}\}_{h\in\mathcal{H}_{\mathrm{src}}}$ of size N; task distributions $\{\mathcal{D}_h\}$; verifier q; mutation operator $\mathcal{M}_\ell$; host weights $\{w_h\}$; fixed coefficients $\lambda_\mu, \lambda_V, \lambda_W, \lambda_M, \lambda_U, \lambda_{\mathrm{cost}}$; feasibility tolerance ϵ; elite count K; migration interval τ; migration limit K_{mig}; predefined generation count T

Ensure: Final source populations $\{\mathcal{P}_h^{(T)}\}$ and feasible nondominated archive $\mathcal{Q}$

Initialize $\mathcal{Q} \leftarrow \varnothing$
Initialize the visited-destination set $\mathcal{M}_h(\mathbf{s}) \leftarrow \varnothing$ for every initial candidate $\mathbf{s} \in \mathcal{P}_h^{(0)}$
for $t = 0, \ldots, T-1$ **do**
  for all $h \in \mathcal{H}_{\mathrm{src}}$ **do**
    Sample the generation minibatch $\mathcal{B}_h^{(t)} \subset \mathcal{D}_h$
    Evaluate each $\mathbf{s} \in \mathcal{P}_h^{(t)}$ on $\mathcal{B}_h^{(t)}$ with and without the skill, and compute $f_h^{\mathrm{local}}(\mathbf{s})$
    Select parents from $\mathcal{P}_h^{(t)}$ according to host-local fitness, reproductive fitness when available, and cumulative reproductive credit when $\mathcal{M}_h(\mathbf{s}) \neq \varnothing$
    Initialize $\mathcal{C}_h^{(t)} \leftarrow \varnothing$
    for all selected parents $\mathbf{s}_h$ **do**
      Select a layer $\ell \in \{\mathrm{core}, \mathrm{adapter}, \mathrm{invocation}\}$ according to the prescribed mutation process
      Generate $\widetilde{\mathbf{s}}_h \leftarrow \mathcal{M}_\ell(\mathbf{s}_h; \mathcal{B}_h, \boldsymbol{\xi})$
      Evaluate $\widetilde{\mathbf{s}}_h$ on $\mathcal{B}_h^{(t)}$ and compute $f_h^{\mathrm{local}}(\widetilde{\mathbf{s}}_h)$
      Add $\widetilde{\mathbf{s}}_h$ to $\mathcal{C}_h^{(t)}$
    end for
    Initialize the incoming migrant set $\mathcal{G}_h^{(t)} \leftarrow \varnothing$
  end for
  Evaluate candidates required for cross-host selection by instantiating their preserved cores on every source host and every probe host
  Compute $\mu(\mathbf{s})$, $V(\mathbf{s})$, $W(\mathbf{s})$, $M(\mathbf{s})$, $U(\mathbf{s})$, $F(\mathbf{s})$, and $\phi(\mathbf{s})$ from the resulting gains
  if $t+1$ is a multiple of τ **then**
    for all $h \in \mathcal{H}_{\mathrm{src}}$ **do**
      Select up to K_{mig} feasible candidates from island $\mathcal{I}_h$ according to $F(\mathbf{s})$
      for all selected $\mathbf{s}_h$ and $h' \in \mathcal{H}_{\mathrm{src}} \setminus \{h\}$ **do**
        Compute the pre-migration destination reference $\overline{g}_{h'}^{(t)}$ from $\mathcal{P}_{h'}^{(t)}$
        Construct $\mathbf{s}_{h\to h'} \leftarrow T_{h\to h'}(\mathbf{s}_h)$ while preserving the core $\mathbf{c}$
        Evaluate $\mathbf{s}_{h\to h'}$ on the held-out destination minibatch and compute $g_{h'}(\mathbf{s}_{h\to h'})$
        Assign $R_{h\to h'}(\mathbf{s}_h) \leftarrow g_{h'}(\mathbf{s}_{h\to h'}) - \max\{0, \overline{g}_{h'}^{(t)}\}$
        Update $\mathcal{M}_h(\mathbf{s}_h) \leftarrow \mathcal{M}_h(\mathbf{s}_h) \cup \{h'\}$ and recompute $\overline{R}_h(\mathbf{s}_h)$
        Add $\mathbf{s}_{h\to h'}$ to $\mathcal{G}_{h'}^{(t)}$
      end for
    end for
  end if
  for all $h \in \mathcal{H}_{\mathrm{src}}$ **do**
    Retain the K elites from $\mathcal{P}_h^{(t)}$, ranking feasible candidates above infeasible candidates
    Fill the remaining $N-K$ positions with candidates from $\mathcal{C}_h^{(t)} \cup \mathcal{G}_h^{(t)}$ according to f_h^{local}, producing $\mathcal{P}_h^{(t+1)}$
  end for
  Add feasible evaluated candidates to $\mathcal{Q}$
  Remove from $\mathcal{Q}$ every candidate dominated with respect to $\big(\mu, -V, W, M, U\big)$
end for
return $\{\mathcal{P}_h^{(T)}\}_{h\in\mathcal{H}_{\mathrm{src}}}$ and $\mathcal{Q}$

or exhaustion of the execution budget. Skills are loaded before the task instruction and consist of a platform-agnostic core, a model-specific adapter, and a harness-specific invocation layer.

B.2 Preprocessing and Evaluation

We reset every task environment to an identical snapshot before each execution. We canonicalize tool outputs by sorting JSON keys, normalizing path separators, removing nondeterministic timestamps and temporary identifiers, and truncating each observation to 8,192 tokens. We decode text files as UTF-8 and replace invalid byte sequences. We compare tabular values after stripping whitespace and applying type-aware parsing. Numeric outputs are normalized to IEEE double precision and accepted within an absolute tolerance of 10^{-6} or a relative tolerance of 10^{-4}. String comparisons apply Unicode NFKC normalization, line-ending normalization, and trailing-whitespace removal. When a task admits multiple valid artifacts or answers, we count a prediction as correct if it satisfies any verifier-approved target specification.

The primary metric is verifier-backed task pass rate. A task receives a score of one only when all mandatory functional checks pass, including unit tests, artifact-existence checks, schema constraints, and required state changes. Otherwise, it receives a score of zero. For partially decomposable tasks, we retain component scores for mutation feedback but exclude them from the reported pass rate. We compute host-specific skill gain as the paired difference between the pass rate with the skill and the no-skill pass rate under the same task and seed. We also report the mean pass rate across hosts, the weak-host pass rate, the minimum host-level pass rate, and the standard deviation of host-level pass rates. We normalize execution cost by the total model-token and tool-execution cost of Single-Host Evolution.

We evaluate all stochastic methods with five paired evolutionary seeds, $\{13, 21, 42, 87, 100\}$. Each seed fixes the task order, initial population, mutation sampling stream, and per-host decoding stream. Tables report the arithmetic mean and sample standard deviation across seeds. Final evaluations use greedy decoding with temperature 0. Mutation generation uses temperature 0.7, top-$p = 0.95$, and at most 4,096 new tokens. We evaluate each final skill once per task and seed, without majority voting or test-time best-of-N selection. Statistical comparisons use paired bootstrap resampling over test tasks with 10,000 replicates and a two-sided significance threshold of $p < 0.05$.

We compare CMRF with No Skill, Best-of-N, Single-Host Evolution, Pooled Average, Worst-Case Selection, and Pareto Search. Best-of-N samples independent candidates and selects the candidate with the highest development pass rate. Single-Host Evolution optimizes only the Qwen2.5-72B-Instruct ReAct host. Pooled Average ranks candidates by uniformly averaged source-host gain. Worst-Case Selection optimizes the minimum source-host gain. Pareto Search maintains candidates that are nondominated in mean pass rate and normalized execution cost. All evolutionary baselines use the same mutation operators, initialization, development tasks, and global budget of 768 candidate–host evaluations.

Before scoring, we repair malformed tool calls only through deterministic JSON parsing rules, including Markdown-fence removal and extraction of the first complete JSON object. We treat calls with missing required arguments, unknown tools, or type-incompatible values as execution failures. Timeouts, sandbox violations, and uncaught runtime exceptions receive a score of zero. We retry infrastructure failures unrelated to agent output once with the same decoding seed and exclude the execution if the retry also fails.

B.3 Implementation and Hardware

We serve all open-weight models locally with vLLM using bfloat16 weights, tensor parallelism, paged attention, and a fixed context window of 32,768 tokens. Qwen2.5-72B-Instruct serves as the strong source execution model. It also generates layer-wise mutations, critiques failed trajectories, and constructs destination-specific adapters. Qwen2.5-7B-Instruct serves as the weak source execution model. Llama-3.1-70B-Instruct and Mistral-7B-Instruct-v0.3 serve exclusively as held-out execution models and do not produce mutations. Each host uses the same assigned checkpoint throughout evolution and final evaluation. We do not update model parameters or use external API models.

The two source harnesses use the same sandbox and tools but implement different control loops and invocation schemas. The ReAct harness alternates between one model response and one tool execution and exposes tools through JSON schemas. The plan–execute–verify harness first requests a bounded plan, executes tool calls through a dispatcher, and inserts an explicit artifact-verification step before termination. The unseen harness uses native function-call messages, different argument

names, and a separate artifact registry. During transfer, the platform-agnostic core remains frozen, while the adapter and invocation files are instantiated from fixed host templates. Probe hosts receive no task-specific edits or additional evolution.

CMRF uses four source-host islands, each with a population of eight. Each island retains two elites and proposes six mutated offspring per generation. Mutation targets the core, adapter, and invocation layers with probabilities 0.40, 0.30, and 0.30, respectively. Each mutation can modify at most one layer and 25% of the tokens in that layer. We reject candidates that exceed 12,000 skill tokens, introduce unavailable dependencies, or fail static tool-schema validation. A lightweight proxy verifier applies syntax checks, interface-compatibility checks, and replay of previously failing steps, removing approximately 58% of invalid proposals before full host execution.

Evolution runs for at most 24 generations and stops when the shared ledger reaches 768 candidate–host evaluations. Every local evaluation, migration evaluation, and cross-host confirmation consumes one ledger entry. Parent sampling uses rank-based selection with probability proportional to $\exp(-r/2)$, where r denotes the candidate rank. We remove duplicate genomes using normalized SHA-256 hashes. Migration occurs every four generations. Each island sends its two highest-ranked feasible candidates to the other source islands in round-robin order. At the destination, the core is preserved, the adapter and invocation layers are regenerated, and the migrant is evaluated on a held-out four-task minibatch. A migrant receives positive reproductive credit only when its destination gain exceeds zero and the destination population's pre-migration mean gain.

We use uniform source-host weights and define weak hosts as the bottom half of hosts ranked by no-skill development pass rate, corresponding to $\rho = 0.5$. The reproductive-fitness coefficients are fixed on the development split:

$$(\lambda_\mu, \lambda_V, \lambda_W, \lambda_M, \lambda_U) = (1.0,\ 0.5,\ 0.5,\ 0.75,\ 0.25).$$

A candidate is feasible when its gain on every source host is at least $-\epsilon$, where $\epsilon = 0.02$. We add reproductive credit to the cross-host fitness with coefficient 0.5. Local selection uses gain-only scoring, $\lambda_{\text{cost}} = 0$, because all compared methods have matched candidate-evaluation budgets. The returned archive contains candidates that are nondominated with respect to mean gain, negative host variance, weak-host gain, worst-host gain, and probe transferability. We select one deployment skill from this archive using reproductive fitness computed only on the development split.

Each task runs in an isolated Docker container with 8 virtual CPU cores, 32 GB of RAM, a 20 GB writable workspace, disabled outbound networking, and a 20-minute wall-clock timeout. Model inference is distributed across eight NVIDIA A100 80 GB GPUs. The 72B and 70B models use four-way tensor parallelism, while the 7B models use one GPU. Independent islands and verifier containers run asynchronously. A complete five-seed experimental run requires approximately 29 aggregate GPU-hours per seed and 8.4 hours of wall-clock time under parallel island evaluation. The implementation uses Python 3.11, PyTorch 2.4, CUDA 12.1, vLLM 0.6, Transformers 4.45, and Docker 27. We set random seeds for Python, NumPy, PyTorch, and the inference scheduler. All prompts, host templates, task manifests, verifier versions, and environment images are pinned by content hash.

C Additional Experimental Results

C.1 Performance by Task Category

Table 4 reports performance across four task categories. CMRF achieves the highest pass rate in every category. Compared with the strongest baseline in each category, CMRF improves the pass rate by 1.8 percentage points for software engineering, 1.8 points for data analysis, 2.9 points for document processing, and 4.3 points for structured web interaction.

C.2 Evolution-Budget Scaling

Table 5 reports performance under different candidate-evaluation budgets. At 192 evaluations, CMRF exceeds Pooled Average by 1.3 percentage points on the source-host mean and 1.5 points on the unseen-host average. At 1,024 evaluations, the corresponding differences are 2.8 and 3.6 percentage points. For CMRF, increasing the budget from 768 to 1,024 evaluations changes the source-host mean from 48.6% to 49.2% and the unseen-host average from 42.5% to 43.1%.

Table 4: Pass rate (%) by task category on the source hosts. Values are mean ± standard deviation over five evolutionary seeds. Each category contains 12 tasks.

Method	Software Engineering	Data Analysis	Document Processing	Structured Web	Average
No Skill	35.8±1.2	34.1±1.0	30.2±1.3	27.5±1.4	31.9±0.8
Best-of-N	45.0±1.7	42.8±1.5	39.3±1.8	36.1±1.9	40.8±1.3
Single-Host Evolution	48.6±1.5	46.2±1.4	41.9±1.6	38.1±1.8	43.7±1.1
Worst-Case Selection	48.7±1.3	45.9±1.2	43.7±1.5	41.3±1.6	44.9±1.0
Pooled Average	50.3±1.2	47.4±1.1	44.6±1.3	40.9±1.5	45.8±0.9
CMRF (ours)	**52.1±1.0**	**49.2±1.1**	**47.5±1.2**	**45.6±1.3**	**48.6±0.8**

Table 5: Source-host and unseen-host performance as a function of the candidate-evaluation budget. Values are mean ± standard deviation over five seeds. All methods use identical budgets at each operating point.

Evaluations	Method	Source Mean	Unseen Average
192	Single-Host Evolution	39.8±1.5	31.7±1.8
	Pooled Average	41.2±1.3	34.6±1.5
	CMRF (ours)	**42.5±1.2**	**36.1±1.4**
384	Single-Host Evolution	41.9±1.3	33.2±1.6
	Pooled Average	43.6±1.1	36.8±1.2
	CMRF (ours)	**45.4±1.0**	**39.2±1.1**
768	Single-Host Evolution	43.7±1.1	34.8±1.4
	Pooled Average	45.8±0.9	38.9±0.9
	CMRF (ours)	**48.6±0.8**	**42.5±0.8**
1,024	Single-Host Evolution	44.2±1.0	35.1±1.3
	Pooled Average	46.4±0.9	39.5±1.0
	CMRF (ours)	**49.2±0.8**	**43.1±0.9**

C.3 Migration Schedule

Table 6 reports performance for different migration intervals under a fixed candidate-evaluation budget. Migration every four generations achieves a source-host mean of 48.6%, weak-host performance of 41.5%, an unseen-host average of 42.5%, and unseen-host worst-case performance of 38.4%. Migration every generation has an execution cost of 2.92×. Migration every eight generations has an execution cost of 2.66× and produces an unseen-host average of 41.3%.

C.4 Model and Harness Shift Decomposition

Table 7 separates transfer degradation associated with model shifts, harness shifts, and their interaction. Under harness shifts, the monolithic-skill variant achieves a pass rate of 38.7%, compared with 44.3% for CMRF. Removing migration changes performance under model shifts from 43.6% to 41.5%. When both host components are unseen, CMRF achieves a pass rate of 38.4% and an interaction gap of 5.6 percentage points.

C.5 Transfer by Source-Host Capability

Table 8 groups destination hosts by their unassisted pass rate. CMRF exceeds Pooled Average by 4.7 percentage points on weak hosts, 3.6 points on medium hosts, and 3.1 points on strong hosts. The corresponding normalized gains for CMRF are 20.9%, 20.2%, and 18.6%.

C.6 Robustness Across Evolutionary Seeds

Table 9 reports results for each evolutionary seed. CMRF achieves the highest source-host mean and unseen-host average for all five seeds. Its ranges are 1.8 percentage points for both metrics, compared with 1.9 and 2.2 points for Pooled Average and 2.4 and 3.4 points for Single-Host Evolution.

Table 6: Sensitivity to the migration interval. The interval denotes the number of generations between migration events. Values are mean ± standard deviation over five seeds.

Migration Interval	Source Mean ↑	Weak Host ↑	Unseen Avg. ↑	Unseen Worst ↑	Cost ↓
No migration	46.2±1.0	37.8±1.3	39.1±1.1	34.6±1.5	2.63×
Every generation	47.3±1.0	39.6±1.2	40.8±1.0	36.7±1.4	2.92×
Every 2 generations	48.0±0.9	40.7±1.1	41.7±0.9	37.6±1.3	2.80×
Every 4 generations	**48.6±0.8**	**41.5±1.0**	**42.5±0.8**	**38.4±1.3**	2.71×
Every 8 generations	47.7±0.9	40.1±1.2	41.3±1.0	36.9±1.4	**2.66×**

Table 7: Pass rate (%) under controlled host shifts. Values are mean ± standard deviation over five seeds. "Both unseen" changes the model and harness simultaneously.

Variant	Model Shift	Harness Shift	Both Unseen	Interaction Gap ↓
Single-Host Evolution	36.9±1.4	36.7±1.6	28.7±1.9	8.1±1.2
Pooled Average	40.6±1.1	40.8±1.1	33.8±1.6	6.9±1.0
CMRF w/o migration	41.5±1.2	41.2±1.3	34.6±1.5	6.8±1.1
CMRF with monolithic skill	40.9±1.3	38.7±1.5	33.8±1.7	6.0±1.2
CMRF (ours)	**43.6±0.9**	**44.3±0.9**	**38.4±1.3**	**5.6±0.9**

C.7 Task-Level Consistency

Table 10 compares CMRF with each baseline across the 16 held-out test tasks. Against Pooled Average and Worst-Case Selection, CMRF records 12 wins, three ties, and one loss under the one-percentage-point threshold. The median differences are $+2.5$ percentage points against Pooled Average and $+3.4$ percentage points against Worst-Case Selection.

D Prompt Templates

D.1 Overview

This appendix specifies the prompt templates used to construct, execute, mutate, and transfer skills across model–harness hosts. The templates separate platform-agnostic procedures, model-specific instructions, and harness-specific invocation semantics. All methods use the same task instructions, tool descriptions, context limits, and output constraints unless stated otherwise. Methods differ in candidate generation, skill representation, fitness-based selection, and migration; the underlying task prompt remains fixed.

D.2 Notation

We use the following placeholders:

- {`TASK`}: the natural-language task instruction.
- {`CONTEXT`}: task-specific context, including input documents, repository state, web state, or structured data.
- {`ARTIFACTS`}: available files, intermediate artifacts, and persistent state.
- {`TOOL_SPECIFICATIONS`}: the tools exposed by the current harness, including names, argument schemas, return schemas, and usage constraints.
- {`EXECUTION_BUDGET`}: the maximum number of model steps, tool calls, tokens, or other bounded resources.
- {`CORE`}: the platform-agnostic skill core containing reusable procedures, checks, and validation rules.
- {`ADAPTER`}: model-specific instructions for executing the core.
- {`INVOCATION`}: harness-specific instructions for skill loading and tool invocation.

Table 8: Transfer performance grouped by destination-host capability. Host groups are defined using tertiles of no-skill validation performance. Absolute gain is measured in percentage points, and normalized gain is computed relative to the remaining headroom. Values are mean ± standard deviation over five seeds.

Destination Group	No Skill	Single-Host Evolution	Pooled Average	CMRF (ours)	CMRF Normalized Gain
Weak hosts	23.9±1.2	31.0±1.7	35.1±1.4	**39.8±1.2**	**20.9±1.8**
Medium hosts	28.7±1.0	35.8±1.5	39.5±1.1	**43.1±1.0**	**20.2±1.5**
Strong hosts	34.8±0.9	40.9±1.3	43.8±1.0	**46.9±0.9**	**18.6±1.4**

Table 9: Per-seed pass rates (%) for the source-host mean and unseen-host average. The final columns report the mean, standard deviation, and range over five seeds.

Metric	Method	Seed 1	Seed 2	Seed 3	Seed 4	Seed 5	Mean	Std.	Range
Source Mean	Single-Host Evolution	42.4	44.8	43.2	44.5	43.6	43.7	1.0	2.4
	Pooled Average	44.7	46.6	45.5	46.3	45.9	45.8	0.7	1.9
	CMRF (ours)	47.5	49.3	48.2	49.0	49.0	48.6	0.7	1.8
Unseen Average	Single-Host Evolution	33.0	36.4	34.1	35.9	34.6	34.8	1.4	3.4
	Pooled Average	37.8	40.0	38.5	39.6	38.6	38.9	0.9	2.2
	CMRF (ours)	41.4	43.2	42.0	43.0	42.9	42.5	0.8	1.8

- {SKILL}: a complete skill, represented as either the tuple ({CORE}, {ADAPTER}, {INVOCATION}) or a monolithic skill in the corresponding ablation.
- {EXECUTION_TRACE}: a trajectory containing model outputs, tool calls, tool returns, intermediate artifacts, and termination status.
- {VERIFIER_FEEDBACK}: verifier outcomes and task-level failure information available to the optimizer.
- {HOST_FEEDBACK}: accumulated execution feedback for a specific model–harness host.
- {SOURCE_HOST} and {DESTINATION_HOST}: source and destination host descriptions, including model characteristics and harness interfaces.
- {LAYER}: the mutation target, one of `core`, `adapter`, or `invocation`.
- {CANDIDATE_ID}: a unique identifier assigned to a generated skill candidate.

Text enclosed in braces is replaced at runtime. Tool calls use the native function-calling or action interface of the corresponding harness and must conform exactly to {TOOL_SPECIFICATIONS}.

D.3 Main Inference Prompt

Purpose. Execute a task with a host-instantiated skill while satisfying the current harness interface and execution budget.

Inputs. {TASK}, {CONTEXT}, {ARTIFACTS}, {TOOL_SPECIFICATIONS}, {EXECUTION_BUDGET}, {CORE}, {ADAPTER}, and {INVOCATION}.

Expected Output. A sequence of valid harness actions followed by the task artifact or final response required by {TASK}. The final response must not expose private reasoning.

```
You are an agent operating within a bounded tool-use environment.

TASK
{TASK}
```

Table 10: Task-level comparisons on the 16 held-out test tasks. A win or loss requires an absolute pass-rate difference of at least one percentage point; smaller differences are counted as ties. Mean difference is measured in percentage points.

Comparison	Wins	Ties	Losses	Mean Difference	Median Difference
CMRF vs. Best-of-N	14	1	1	$+7.8$	$+7.4$
CMRF vs. Single-Host Evolution	13	2	1	$+4.9$	$+4.6$
CMRF vs. Worst-Case Selection	12	3	1	$+3.7$	$+3.4$
CMRF vs. Pooled Average	12	3	1	$+2.8$	$+2.5$

```
CONTEXT
{CONTEXT}

AVAILABLE ARTIFACTS
{ARTIFACTS}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

EXECUTION BUDGET
{EXECUTION_BUDGET}

TRANSFERABLE SKILL CORE
{CORE}

MODEL ADAPTER
{ADAPTER}

HARNESS INVOCATION LAYER
{INVOCATION}

Use the transferable skill core as the primary procedure. Use the model
adapter to translate this procedure into decisions. Use the harness
invocation layer to load resources and call tools.

Requirements:
1. Decompose the task according to the skill core.
2. Use only tools listed in the tool specifications.
3. Match every tool name, argument name, argument type, and required field
   exactly.
4. Inspect tool returns and generated artifacts before proceeding.
5. Apply the validation and recovery checks defined by the skill.
6. Keep recovery within the execution budget.
7. Claim completion only after verifying the required artifact or final
   result.
8. Return only the requested deliverable and any concise status information
   required by the harness.

Begin task execution.
```

D.4 Baseline Prompts

D.4.1 No-Skill Inference

Purpose. Measure the capability of a host without an external skill.

Inputs. {TASK}, {CONTEXT}, {ARTIFACTS}, {TOOL_SPECIFICATIONS}, and {EXECUTION_BUDGET}.

Expected Output. A sequence of valid harness actions followed by the requested task deliverable.

```
You are an agent operating within a bounded tool-use environment.

TASK
{TASK}

CONTEXT
{CONTEXT}

AVAILABLE ARTIFACTS
{ARTIFACTS}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

EXECUTION BUDGET
{EXECUTION_BUDGET}

Complete the task using the available context, artifacts, and tools.

Requirements:
1. Use only tools listed in the tool specifications.
2. Match every tool name, argument name, argument type, and required field
   exactly.
3. Inspect tool returns and generated artifacts before proceeding.
4. Keep all retries and recovery actions within the execution budget.
5. Claim completion only after producing the requested result.
6. Return only the requested deliverable and any concise status information
   required by the harness.

Begin task execution.
```

D.4.2 BASELINE SKILL GENERATION

Purpose. Generate candidate skills for Best-of-N, Single-Host Evolution, Pooled Average, Worst-Case Selection, and Pareto Search. These baselines use the same candidate-generation prompt and differ only in candidate evaluation and selection.

Inputs. {TASK_DISTRIBUTION}, {TOOL_SPECIFICATIONS}, {HOST_DESCRIPTION}, and {OBSERVED_FAILURES}.

Expected Output. A self-contained candidate skill in the structured format below.

```
Construct a reusable agent skill for the following task distribution.

TASK DISTRIBUTION
{TASK_DISTRIBUTION}

TARGET HOST
{HOST_DESCRIPTION}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

OBSERVED FAILURE MODES
{OBSERVED_FAILURES}

The skill must define an executable, reusable procedure for the task
distribution. It must not solve a single task instance. Include task
decomposition, action-selection rules, intermediate checks, bounded recovery
procedures, and final validation. Exclude task-specific answers and
```

```
hard-coded instance values.

Return exactly the following structure:

<skill>
  <procedure>
    A numbered end-to-end procedure.
  </procedure>
  <tool_usage>
    Rules for selecting tools, constructing arguments, and interpreting
    tool returns.
  </tool_usage>
  <verification>
    Intermediate checkpoints and final completion criteria.
  </verification>
  <recovery>
    Bounded responses to anticipated failures.
  </recovery>
</skill>
```

D.5 Reasoning and Rationale Prompts

D.5.1 Execution Diagnosis

Purpose. Convert execution traces and verifier outcomes into concise mutation feedback without modifying the candidate skill.

Inputs. {TASK}, {SKILL}, {EXECUTION_TRACE}, {VERIFIER_FEEDBACK}, and {TOOL_SPECIFICATIONS}.

Expected Output. A JSON object identifying the earliest consequential failure, the affected skill layer, and a bounded repair recommendation.

```
Analyze the following skill execution.

TASK
{TASK}

SKILL
{SKILL}

EXECUTION TRACE
{EXECUTION_TRACE}

VERIFIER FEEDBACK
{VERIFIER_FEEDBACK}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

Identify the earliest consequential failure. Classify it as:
- core: task decomposition, general procedure, checking, or recovery;
- adapter: model-facing instruction, ambiguity, ordering, or decision rule;
- invocation: skill loading, tool selection, argument schema, return parsing,
  path handling, or artifact exposure;
- none: the evidence supports no actionable skill defect.

Use only evidence from the trace and verifier feedback. Do not rewrite the
skill.

Return valid JSON only:
```

```
{
  "failure_stage": "<short stage name>",
  "failure_summary": "<concise evidence-based description>",
  "affected_layer": "core|adapter|invocation|none",
  "repair_objective": "<single bounded repair objective>",
  "preserve": [
    "<behavior that should remain unchanged>"
  ],
  "evidence": [
    "<trace or verifier observation>"
  ]
}
```

D.5.2 LAYER-WISE SKILL MUTATION

Purpose. Generate an offspring skill by modifying exactly one genome layer based on accumulated host feedback.

Inputs. {CORE}, {ADAPTER}, {INVOCATION}, {LAYER}, {HOST_FEEDBACK}, {HOST_DESCRIPTION}, and {TOOL_SPECIFICATIONS}.

Expected Output. A complete structured skill in which the non-target layers are reproduced verbatim and the target layer contains a minimal, feedback-grounded revision.

```
Revise one layer of the following structured agent skill.

TARGET LAYER
{LAYER}

HOST
{HOST_DESCRIPTION}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

ACCUMULATED HOST FEEDBACK
{HOST_FEEDBACK}

CURRENT CORE
{CORE}

CURRENT ADAPTER
{ADAPTER}

CURRENT INVOCATION LAYER
{INVOCATION}

Mutation rules:
1. Modify only the target layer.
2. Reproduce both non-target layers verbatim.
3. Make the smallest change that addresses the supported failure modes.
4. Preserve successful procedures and validation checks.
5. Do not add task-instance answers, unsupported tools, or host interfaces
   absent from the tool specifications.
6. Core revisions must remain platform-agnostic and must not contain model
   names, harness commands, tool schemas, wrapper syntax, or environment-
   specific paths.
7. Adapter revisions can change model-facing decomposition, explicitness,
   ordering, checkpoints, and recovery guidance, but must not redefine tool
   schemas.
8. Invocation revisions can repair loading, tool mapping, argument
```

```
   construction, return parsing, paths, and artifact exposure, but must not
   change the general task procedure.

Return exactly:
<skill candidate_id="{CANDIDATE_ID}">
  <core>
  [complete core]
  </core>
  <adapter>
  [complete adapter]
  </adapter>
  <invocation>
  [complete invocation layer]
  </invocation>
</skill>
```

D.6 Data Generation and Candidate Construction Prompts

D.6.1 Initial Structured Skill Construction

Purpose. Create an initial structured skill genome before host-island evolution.

Inputs. {TASK_DISTRIBUTION}, {HOST_DESCRIPTION}, {TOOL_SPECIFICATIONS}, and {EXECUTION_BUDGET}.

Expected Output. A three-layer skill containing a platform-agnostic core, a model adapter, and a harness invocation layer.

```
Construct a reusable skill for an agent that solves tasks from the following
distribution.

TASK DISTRIBUTION
{TASK_DISTRIBUTION}

HOST
{HOST_DESCRIPTION}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

EXECUTION BUDGET
{EXECUTION_BUDGET}

Represent the skill using three layers.

The core must contain:
- a reusable task-decomposition procedure;
- abstract operations independent of any particular tool schema;
- intermediate checkpoints;
- bounded failure recovery;
- final validation rules.

The model adapter must:
- translate the core into instructions for the target execution model;
- make decision points and stopping conditions explicit;
- reduce unnecessary planning while preserving the core procedure.

The harness invocation layer must:
- map abstract operations to available tool names;
- specify exact handling of arguments and return fields;
- describe skill loading, artifact access, and path conventions;
```

```
- include interface-level checks for malformed or mismatched calls.

Exclude model names, harness commands, concrete tool schemas, wrapper syntax,
and environment-specific paths from the core. Do not encode answers to
individual task instances.

Return exactly:
<skill candidate_id="{CANDIDATE_ID}">
  <core>
  [platform-agnostic procedure]
  </core>
  <adapter>
  [model-specific execution instructions]
  </adapter>
  <invocation>
  [harness-specific loading and tool mapping]
  </invocation>
</skill>
```

D.6.2 DESTINATION-HOST ADAPTER CONSTRUCTION

Purpose. Instantiate a migrated or frozen skill on a destination model while preserving its transferable core.

Inputs. {CORE}, {SOURCE_ADAPTER}, {SOURCE_HOST}, and {DESTINATION_HOST}.

Expected Output. A destination-specific model adapter. The core remains unchanged.

```
Construct a model adapter for a transferred skill.

TRANSFERABLE CORE
{CORE}

SOURCE HOST
{SOURCE_HOST}

SOURCE ADAPTER
{SOURCE_ADAPTER}

DESTINATION HOST
{DESTINATION_HOST}

Translate the core into concise execution instructions for the destination
model.

Requirements:
1. Preserve every procedural dependency, checkpoint, recovery condition, and
   completion criterion in the core.
2. Adapt only model-facing properties, including instruction explicitness,
   decomposition granularity, action ordering, state tracking, and stopping
   criteria.
3. Do not add harness syntax, tool names, argument schemas, file paths, or
   task-instance content.
4. Do not modify the core or restate it as a new procedure.
5. Use explicit bounded instructions when the destination model requires
   additional execution guidance.

Return exactly:
<adapter>
[destination-specific model adapter]
</adapter>
```

D.6.3 DESTINATION-HARNESS INVOCATION CONSTRUCTION

Purpose. Construct the invocation layer required to execute a transferred core under a destination harness.

Inputs. {CORE}, {SOURCE_INVOCATION}, {SOURCE_HOST}, {DESTINATION_HOST}, and {TOOL_SPECIFICATIONS}.

Expected Output. A destination-specific invocation layer that maps abstract operations to the destination harness.

```
Construct a harness invocation layer for a transferred skill.

TRANSFERABLE CORE
{CORE}

SOURCE HOST
{SOURCE_HOST}

SOURCE INVOCATION LAYER
{SOURCE_INVOCATION}

DESTINATION HOST
{DESTINATION_HOST}

DESTINATION TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

Map the abstract operations in the core to the destination harness.

Requirements:
1. Preserve the core without modification.
2. Use only tools and interfaces present in the destination specifications.
3. Match tool names, argument names, argument types, required fields, and
   return schemas exactly.
4. Specify skill-loading behavior, artifact-access conventions, path
   handling, and result exposure where applicable.
5. Include checks for missing arguments, schema mismatches, malformed tool
   returns, and unavailable artifacts.
6. Do not introduce new procedural goals or task-instance content.
7. Mark an abstract operation as unsupported if no valid destination mapping
   exists. Do not invent an interface.

Return exactly:
<invocation>
  <loading>
  [destination skill-loading instructions]
  </loading>
  <tool_mapping>
  [abstract-operation to tool-interface mappings]
  </tool_mapping>
  <return_handling>
  [tool-return and artifact-handling rules]
  </return_handling>
  <interface_checks>
  [schema and availability checks]
  </interface_checks>
</invocation>
```

D.6.4 PROXY PRE-SCREENING

Purpose. Reject malformed, unsupported, or clearly non-transferable mutations before full host execution.

Inputs. {PARENT_SKILL}, {CANDIDATE_SKILL}, {LAYER}, {HOST_DESCRIPTION}, and {TOOL_SPECIFICATIONS}.

Expected Output. A JSON decision indicating whether the candidate is eligible for full evaluation.

```
Screen the proposed skill mutation before execution.

HOST
{HOST_DESCRIPTION}

TARGET LAYER
{LAYER}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

PARENT SKILL
{PARENT_SKILL}

CANDIDATE SKILL
{CANDIDATE_SKILL}

Reject the candidate if any of the following conditions holds:
- the output is malformed or incomplete;
- a non-target layer differs from the parent;
- the core contains host-specific commands, tool schemas, wrappers, or paths;
- the adapter redefines tool interfaces;
- the invocation layer uses unavailable tools or invalid arguments;
- required checks or completion criteria were removed without supporting
  evidence;
- the candidate contains task-instance answers or hard-coded instance values;
- the revision does not address the stated mutation target;
- the candidate introduces an unbounded loop or recovery procedure.

Return valid JSON only:
{
  "decision": "accept|reject",
  "violations": [
    "<specific violation>"
  ],
  "target_layer_changed": true,
  "non_target_layers_preserved": true,
  "interface_compatible": true
}
```

D.7 EVALUATION PROMPTS

Task-specific executable verifiers determine task success. Each verifier computes a bounded score $q(h, x, s) \in [0, 1]$ from the produced artifact, final environment state, or structured task output. The reported experiments therefore use no natural-language evaluation or pairwise-preference prompt.

D.8 Ablation Prompt Variants

D.8.1 Monolithic Skill Mutation

Purpose. Implement the monolithic-skill ablation by revising a single undifferentiated skill representation.

Inputs. {MONOLITHIC_SKILL}, {HOST_FEEDBACK}, {HOST_DESCRIPTION}, and {TOOL_SPECIFICATIONS}.

Expected Output. A complete revised monolithic skill.

```
Revise the following monolithic agent skill using the accumulated host
feedback.

HOST
{HOST_DESCRIPTION}

TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

ACCUMULATED HOST FEEDBACK
{HOST_FEEDBACK}

CURRENT SKILL
{MONOLITHIC_SKILL}

Produce a complete revised skill that improves task execution on the target
host. The revision can jointly modify procedural logic, model-facing
instructions, and harness invocation details.

Requirements:
1. Address only failure modes supported by the feedback.
2. Preserve successful behavior when compatible with the revision.
3. Use only tools and interfaces listed in the tool specifications.
4. Include task decomposition, tool-use rules, checkpoints, bounded recovery,
   and completion criteria.
5. Do not include task-instance answers or hard-coded instance values.

Return exactly:
<skill candidate_id="{CANDIDATE_ID}">
[complete monolithic skill]
</skill>
```

D.8.2 Migration Without Layer Reconstruction

Purpose. Define the transfer form used when the layered representation is disabled. Destination-host adaptation receives the complete monolithic skill.

Inputs. {MONOLITHIC_SKILL}, {SOURCE_HOST}, {DESTINATION_HOST}, and {TOOL_SPECIFICATIONS}.

Expected Output. A destination-compatible monolithic skill.

```
Adapt the following monolithic skill from a source host to a destination host.

SOURCE HOST
{SOURCE_HOST}

DESTINATION HOST
```

```
{DESTINATION_HOST}

DESTINATION TOOL SPECIFICATIONS
{TOOL_SPECIFICATIONS}

SOURCE SKILL
{MONOLITHIC_SKILL}

Revise the skill for execution under the destination model and harness.
Preserve compatible reusable behavior and repair all model-facing and
interface-specific incompatibilities.

Requirements:
1. Use only destination tools and schemas.
2. Preserve task decomposition, verification, and bounded recovery when they
   are compatible with the destination host.
3. Replace invalid tool names, arguments, paths, loading commands, and
   return-field assumptions.
4. Do not include task-instance answers or unsupported interfaces.

Return exactly:
<skill candidate_id="{CANDIDATE_ID}">
[destination-compatible monolithic skill]
</skill>
```

All fitness-term ablations use the full method's generation, mutation, transfer, and inference prompts. Removing island migration disables destination-host adapter and invocation construction during evolution. Removing the worst-host term, variance penalty, or weak-host credit changes only the candidate-selection score. Changing the migration frequency changes only the schedule for invoking the destination-host construction templates.

D.9 Output Formatting

Structured skill outputs must be valid UTF-8 text and preserve the required XML-style delimiters exactly. Diagnostic and screening outputs must be valid JSON without Markdown fences, comments, trailing commas, or additional prose. Tool calls must use the native structured-action representation of the active harness and conform exactly to the supplied tool schema.

For JSON outputs, strings must escape embedded quotation marks and control characters. An empty list represents missing evidence; the system must not fabricate an explanation. For skill outputs, every required layer must be present even when no revision is made. During layer-wise mutation, the non-target layers must be copied verbatim from the parent candidate. Final task outputs follow the task-specified format and are evaluated directly by the corresponding executable verifier.

E Case Study

We present representative execution traces showing how transfer-aware evolution changes an agent skill's content and behavior. The examples cover two recurrent sources of cross-host failure: harness-level schema mismatches and model-dependent execution of long-horizon artifact workflows. We normalize all identifiers while preserving the procedural structure and failure modes.

E.1 Case I: Recovering from a Tool-Schema Mismatch

Consider a task that retrieves the weather for Boston and returns a short structured summary. In the source harness, the weather tool accepts a single argument named `city`. A skill produced by single-host evolution embeds this source-specific call directly in its procedure:

```
When weather information is requested:
1. Extract the city from the user request.
```

```
2. Call get_weather(city=<extracted city>).
3. Read temperature, conditions, and precipitation from the result.
4. Return a concise answer.
```

On the source host, the resulting action is valid:

```
{"name": "get_weather",
 "arguments": {"city": "Boston"}}
```

After transfer, the destination harness exposes a semantically equivalent tool with a different schema:

```
get_weather(location: string, unit: string) -> object
```

The single-host skill reproduces the source call verbatim:

```
{"name": "get_weather",
 "arguments": {"city": "Boston"}}
```

The destination harness rejects the request because `city` is unrecognized and the required `unit` argument is missing. This failure occurs before weather reasoning begins. The skill identifies the correct operation but encodes its invocation syntax as part of the operation. A stronger execution model can recover the local schema from the tool documentation, whereas a weaker model often repeats the same call or changes the value while retaining the invalid key.

CMRF represents this behavior using three separable layers. The platform-agnostic core specifies the intended operation without committing to a concrete tool signature:

```
Core procedure: weather lookup

Input:
    A place mention and, when available, a requested unit system.

Procedure:
    1. Resolve the place string from the user request.
    2. Determine the requested unit system.
       If the request does not specify one, use the host adapter's
       locale-compatible default.
    3. Invoke the host's weather-lookup operation using its declared schema.
    4. Validate that the returned object contains a usable condition and
       temperature value.
    5. If validation fails, inspect the returned field names once and map
       semantically equivalent fields before producing the answer.

Do not place concrete argument names in the core procedure.
```

The destination model adapter reduces the reconstruction required from the receiving model:

```
Model adapter:

Before issuing the call, copy argument names from the currently exposed
tool signature. Do not reuse argument names remembered from an example.
Construct all required arguments before calling the tool. If the user does
not specify a unit, use "celsius" for this host.
```

The destination invocation layer binds the abstract operation to the local interface:

```
Invocation mapping:

abstract operation: WEATHER_LOOKUP(place, unit_system)
destination tool:   get_weather
argument mapping:
    place       -> location
    unit_system -> unit
```

The transferred skill emits:

```
{"name": "get_weather",
 "arguments": {
     "location": "Boston",
     "unit": "celsius"
 }}
```

Transfer therefore requires neither a more verbose shared procedure nor an enumeration of known harnesses. The transferable content is the invariant decision sequence: resolve the place, determine the unit system, invoke the available weather operation, and validate the response. Only the binding between semantic slots and concrete argument names changes. During migration, replacing `city` with `location` in a monolithic skill repairs the destination call but breaks hosts that retain the original schema. In the layered representation, this repair is confined to the invocation layer, allowing the core to reproduce across both environments.

The validation instruction also changes the agent's behavior after a successful call. Some harnesses return `condition`, whereas others store an equivalent value under `current.summary`. A source-specialized skill interprets the unfamiliar object as missing data. The CMRF skill checks for the expected semantic content and then performs a bounded field mapping. Limiting recovery to one inspection step prevents repeated tool probes after a genuine service failure.

E.2 Case II: Transferring a Spreadsheet-Cleaning Skill to a Weaker Model

The second case concerns an artifact-manipulation task. The agent receives a spreadsheet of transaction records and must create a cleaned workbook. The requested transformations preserve the original data, normalize date cells, remove records with duplicate transaction identifiers, and add a summary sheet. The final artifact must be saved to the path exposed by the harness.

A single-host-evolved skill written by a stronger model contains an underspecified plan:

```
Open the spreadsheet, clean the data, remove duplicates, create a summary,
and save the completed workbook to the output directory. Verify the result
before responding.
```

This instruction delegates several consequential decisions to the execution model. On the source host, the stronger model infers that the workbook must be copied before modification, identifies the transaction identifier as the deduplication key, and reopens the saved file for verification. After transfer to a weaker model, the observed execution is:

```
1. Load the first worksheet.
2. Drop duplicate rows.
3. Convert the date column.
4. Compute totals in memory.
5. Save the active worksheet as cleaned_output.xlsx.
6. Report completion.
```

The output appears plausible but violates the artifact contract in several ways. The model interprets "Drop duplicate rows" as removing rows that are identical across all columns, including cases in which repeated transaction identifiers have different notes or formatting. Saving only the active worksheet discards auxiliary sheets. The model computes the summary but never writes it to the workbook. It also equates a successful save call with a valid deliverable and does not reopen the file. These failures do not reflect an inability to perform individual spreadsheet operations. They occur because the receiving model must reconstruct implicit procedural assumptions across a long action sequence.

A representative CMRF core expresses these assumptions as executable and verifiable steps:

```
Core procedure: clean a tabular workbook

Inputs:
    source_artifact
    destination_artifact
    primary_record_key
    date_fields
    requested_summary

Preconditions:
    - Confirm that the source artifact exists and is readable.
    - Enumerate worksheets before editing.
    - Never overwrite the source artifact unless explicitly requested.

Procedure:
    1. Load the workbook while preserving all worksheets.
    2. Select the data worksheet by matching required column names rather
       than assuming that the first worksheet is the target.
    3. Snapshot:
           worksheet names,
           source column order,
           source row order,
           and the original value of each primary record key.
    4. Normalize only the declared date fields.
       Preserve missing cells as missing values rather than converting them
       to text.
    5. Detect repeated records using primary_record_key.
       Do not substitute whole-row equality for key-based duplication.
    6. Retain the first complete occurrence of each key unless the task
       specifies another policy. Record removed source-row references.
    7. Create or replace a worksheet named "Summary".
       Write the requested aggregations as workbook cells, not only as
       values in the agent's working memory.
    8. Save to destination_artifact.
    9. Close and reopen destination_artifact.
   10. Validate:
           all non-target worksheets remain present;
           required data columns remain present and in their original order;
           retained primary keys are unique;
           date cells use the intended date representation;
           the Summary worksheet exists and contains formulas or values;
           the source artifact remains unchanged.
   11. Return success only after the reopened artifact passes validation.

Bounded recovery:
    - If a required column is absent, stop and report the missing column.
    - If the saved workbook cannot be reopened, retry the save once using
      the same destination path.
    - Do not silently choose a different deduplication key.
```

For the weaker host, the model adapter converts the core into a linear execution discipline:

```
Model adapter for a limited-planning host:

Execute the numbered procedure in order. Keep a checklist with one item per
step. After each artifact operation, record the observed result before
continuing. Do not combine loading, mutation, saving, and validation into a
single generated action. Treat "saved" and "validated" as different states.
If a precondition fails, stop rather than guessing.
```

The adapter preserves the task decisions while changing their presentation to the model. The explicit distinction between *saved* and *validated* prevents premature termination. The weaker model no longer needs to recover an implicit requirement to reopen the workbook after a successful write.

The invocation layer specifies how different harnesses expose the artifact. One harness supplies an operating-system path, whereas another supplies an artifact handle and requires an explicit publication action:

```
Invocation mapping for the destination harness:

LOAD_WORKBOOK(source_artifact):
    resolve the input artifact handle to a readable staged path;
    pass the staged path to the workbook library.

SAVE_WORKBOOK(destination_artifact):
    write to the harness staging directory;
    reopen the staged file for validation;
    publish the validated file through submit_artifact;
    return the published artifact identifier.

Do not place staging paths or submit_artifact calls in the core procedure.
```

This mapping preserves the data-cleaning policy when the destination exposes files through a different interface. In a monolithic skill, a mutation that adds `submit_artifact` can propagate the destination harness's publication semantics into the general procedure. When this variant returns to a path-based harness, the model can attempt to call an unavailable publication tool or withhold the output because the harness produces no artifact identifier. CMRF confines this behavior to the invocation layer.

The resulting execution trace differs from the underspecified baseline:

```
[check] Source artifact resolved and opened.
[check] Worksheets enumerated; target selected by required columns.
[check] Source workbook copied to destination workspace.
[action] Declared date fields normalized.
[action] Repeated transaction identifiers resolved using the specified key.
[action] Summary worksheet written into the workbook.
[action] Destination workbook saved.
[check] Destination workbook reopened successfully.
[check] Original worksheets and column order preserved.
[check] Retained transaction identifiers are unique.
[check] Summary worksheet is present.
[action] Validated artifact published through the destination harness.
```

Each checklist entry corresponds to an externally inspectable state. The checklist also localizes recovery. If the summary sheet is absent after reopening, the agent returns to the summary-writing step without rerunning date normalization and deduplication. This bounded recovery prevents repairs from modifying portions of the workbook that have already passed validation.

E.3 Case III: Destination Credit Rejects Superficial Compatibility

The final trace shows why successful loading is insufficient evidence of transfer. A migrated document-processing skill instructs an agent to extract headings from a PDF, construct a concise outline, and save the result as a text artifact. Its invocation layer is successfully reconstructed for the destination harness, and the skill loads without an interface error. The destination model follows the extraction instruction literally and includes repeated page headers as document headings:

```
Generated outline:
- UNDER REVIEW AS A CONFERENCE PAPER
- ABSTRACT
- UNDER REVIEW AS A CONFERENCE PAPER
- INTRODUCTION
- UNDER REVIEW AS A CONFERENCE PAPER
- RELATED WORK
```

A source-only evaluation can preserve this variant because it invokes the correct parser and produces a syntactically valid artifact. Destination evaluation reveals a semantic weakness in the shared core: the procedure lacks a rule for distinguishing document structure from recurring page furniture. The subsequent core mutation adds an invariant filtering procedure:

```
When constructing a document outline:
1. Collect heading candidates together with page position and recurrence.
2. Mark a candidate as page furniture when substantially the same text
   recurs at a consistent top or bottom position across multiple pages.
3. Exclude marked page furniture from the outline.
4. Preserve a recurring string when it is part of an explicitly numbered
   section title whose surrounding content changes by section.
5. Verify that each retained heading introduces a distinct content span.
```

After migration, the destination output is:

```
Generated outline:
- Abstract
- Introduction
- Related Work
  - Procedural Knowledge, Tool Use, and Skill Acquisition
  - Optimization and Evolution of Language-Model Programs
  - Agent Evaluation, Generalization, and Host Heterogeneity
- Method
  - Problem Formulation and Framework
  - Structured Skill Genome
  - Host-Island Evolution
  - Cross-Model Reproductive Fitness
  - Migration, Reproductive Credit, and Deployment
- Experiments
- Conclusion
```

This repair belongs in the core because recurring headers are a document-level phenomenon independent of the execution model and harness. Once incorporated, every island can inherit it. Parser initialization, artifact-handle resolution, and output publication remain confined to the invocation layer. This example specifies the role of destination-based reproductive credit: a skill receives credit for producing useful behavior in the receiving host, beyond crossing an interface boundary or completing a tool call.

Across these cases, the transferable unit is a procedure with explicit semantic inputs, observable checkpoints, validation conditions, and bounded recovery actions. CMRF preserves these elements in

the shared core while allowing the model-facing execution discipline and harness-facing bindings to change independently. This separation maintains behavioral stability when the receiving model has different planning capacity, the tool schema uses different argument names, or the harness implements a different artifact lifecycle.